An effective method to decide bounded reordering conformance in the distributed test architecture

Trabajo de fin de grado del Doble Grado en Ingeniería Informática - Matemáticas

Miguel Benito Parejo

Department of "Sistemas Informáticos y Computación"
Facultad de Informática
Universidad Complutense de Madrid

September 2018

Document layout with TEXIS v.1.0.

This document is prepared to be printed double-sided.

An effective method to decide bounded reordering conformance in the distributed test architecture

Trabajo fin de grado
Doble Grado en Ingeniería Informática - Matemáticas
Universidad Complutense de Madrid

Directed by
Mercedes García Merayo
Manuel Núñez García

Department of "Sistemas Informáticos y Computación"
Facultad de Informática
Universidad Complutense de Madrid

September 2018

To Juan Antonio and Eduardo

Acknowledgements

I would like to thank my supervisors for their effort, work, help and advice before and during this thesis. I would like to thank these close friends who helped through the toughest moments. I would like to thank the TEXIS creators for this helpful template. Last, but definitely not least, I would like to thank my family for all their unconditional help and support.

Abstract

In the distributed test architecture, the system under test interacts with its environment at multiple physically distributed ports and the local testers at these ports do not synchronise their actions. This presents many challenges and, in particular, apparently incorrect behaviours can be the consequence of an erroneous assumption about the exact order in which actions were performed at different ports. In previous work, it was defined a conformance relation for the distributed test architecture considering the order possibilities and the distance actions are able to be delayed. Basically, the system under test is faulty if we observe a trace σ such that no *enough* admissible reordering of the actions in σ could have been produced by the specification. This notion takes into account both the way of reordering and the amount of changes a trace can receive. In this thesis we implement an algorithm, and provide the theoretical results proving its correctness, to construct a finite automata able to check the whether a system under test conforms to a specification with respect to this implementation relation. Thus, a side result of the thesis is that we slightly extend the theoretical framework for bounded distributed relations.

Key Words: Software testing; Formal methods; Testing in the distributed architecture.

Resumen

Al realizar testing en una arquitectura distribuida, el sistema en pruebas interactúa con su entorno a través de múltiples puertos físicamente distribuidos y los testeadores locales, situados en estos puertos, no sincronizan sus acciones. Esta limitación presenta muchos desafíos que, en particular, pueden llevar a que comportamientos aparentemente incorrectos puedan ser la consecuencia de una suposición errónea sobre el orden exacto en el que se realizaron las acciones en los diferentes puertos. En un trabajo previo, se definió una relación de conformidad para realizar testing en la arquitectura distribuida considerando los posibles reordenamientos y la distancia que las acciones pueden ser retrasadas. Básicamente, el sistema en pruebas es erroneo si observamos una traza σ tal que no hay una reordenación admisible de *suficientes* intercambios de las acciones en σ que pueda ser producida por la especificación. Ello tiene en cuenta tanto la forma de reordenar como la cantidad de cambios que puede recibir una traza. En este trabajo implementamos un algoritmo, y damos los resultados teóricos que demuestran su corrección, para construir un autómata finito capaz de determinar si una implementación es conforme a una especificación respecto a dicha relación de implementación. Un efecto lateral de esta tesis es que ampliamos ligeramente el marco de testing de arquitecturas distribuidas con cotas.

Palabras Clave: Testing de software; Métodos formales; Testing de arquitectura distribuida.

Contents

List of Figures

Chapter 1

Introduction

This chapter is the introduction of this thesis and presents the main concepts behind this work. This thesis takes as initial step recent work on formal testing in the distributed architecture where conformance relations are adapted to take into account *bounded reorderings* (Hierons et al., 2018). The authors developed the idea of setting a bound for the distance an action might be delayed at when being in a distributed system. In this thesis, we extend this work with a testing framework, strongly based on the idea of *product machine*, by providing an effective method to decide the conformance of a System Under Test (SUT) to a specification. The work in this thesis is not restricted to the theoretical framework: the algorithms have been implemented and can be found at `github.com/miguelbpsg/TFG`.

The rest of the chapter is structured as follows. Section 1.1 presents an explanation of the motivations that brought us to the development of this work. Section 1.2 provides our goals. Finally, in Section 1.3 we present our Work Plan and the structure of the rest of the thesis.

1.1 Motivation

Software testing (Ammann y Offutt, 2017; Myers et al., 2011) is the main technique to validate complex software and hardware systems. Essentially, software testing consists in providing inputs to a system, observe the produced outputs and decide whether the observed behaviour is admissible with

respect to the expected one. The expected behavior can be given by an expert tester (this is the usual approach where the tester acts as an *oracle*), by a set of (in)formal requirements or, as we will consider in this thesis, by a formal specification. Traditionally, testing activities have mainly been manual and, as a consequence, slow, costly and error prone. Nevertheless, it is already well known that testing can be *formalized* (Gaudel, 1995) and during the last 20 years formal methods have been successfully applied to the testing process (Binder et al., 2015; Cavalli et al., 2015; Hierons et al., 2009). In particular, the combination of formal methods and testing has allowed test automation. In addition to well established theories and methodologies, there are several tools that allow potential users to apply techniques developed in this area (Marinescu et al., 2015; Shafique y Labiche, 2015).

Usually, testing is considered as a process where a single tester interacts with the SUT. However, many current systems interact with their environment at physically distributed locations (ports) and, in this case, it is usual to place a separate tester at each port. During the last 15 years there has been important research in distributed testing when testing from a formal model written as a finite state machine (Hierons, 2013, 2015; Hierons y Ural, 2008), input-output transition system (Hierons et al., 2008, 2012, 2018), Petri Net (Ponce de León et al., 2013, 2016), or partial-order automaton (Bochmann et al., 2008; Haar et al., 2007).

The previously mentioned approach, placing a tester at each port, is very practical and sometimes it is the only one that can be used but it has a strong drawback: a local tester is only able to observe the events at its port and it might not be possible to reconstruct the whole global sequence of events that occurred. For example, consider a system with two ports, having a tester place at each port. Tester at port 1 observes output $!o_1$ and the tester at port 2 observes $!o_2$. If the local testers pass this information to a *global* entity that has to decide whether the observed behavior is coherent with its expectations, then it will have to face the problem that it does not know whether the *global* trace was $!o_1!o_2$ or $!o_2!o_1$. In addition to the obvious practical issue, we also face some theoretical non-negligible problems. Actually, if we consider a finite model M acting as a specification, the problem of deciding whether an observation is allowed by M is NP-complete (Hierons, 2012) and the problem of deciding whether a finite model N conforms to M (all observations that can be made of N are allowed by M) is undecidable (Hierons, 2010). In contrast, if testing is considered in a non-distributed setting, the first problem can be decided in low-order polynomial time and the second problem is decidable and can be solved in polynomial time if both N and M are deterministic finite automata (or an observable finite state machine or input-output transition

system).

As noted above, in distributed testing the separate testers cannot synchronize and so we cannot know the relative order of events at different ports. This leads to a notion of observational equivalence in which traces σ_1 and σ_2 are equivalent if and only if for every port p we have that σ_1 and σ_2 have the same projections at p. Under this, $!o_1!o_2$ and $!o_2!o_1$ are observationally equivalent and it should be indifferent whether the SUT was producing the first sequence or the second one. Actually, for every natural number m we have that $!o_1^m!o_2$ and $!o_2!o_1^m$ are observationally equivalent. Previous work by the authors of the framework used in this thesis defined implementation relations, and their associated testing frameworks, to realize these ideas (Hierons et al., 2012, 2014). In addition, there are other papers on distributed testing that considered a notion of observational equivalence in which $\sigma_1 \sim \sigma_2$, for traces σ_1 and σ_2, if σ_1 and σ_2 have the same projections at the ports (Bochmann et al., 2008; Cacciari y Rafiq, 1999; Dssouli y Bochmann, 1985, 1986; Haar et al., 2007; Hierons, 2013, 2015; Hierons et al., 2008; Hierons y Ural, 2008; Jard et al., 1998; Khoumsi, 2002; Luo et al., 1993; Nguyen et al., 2014; Ponce de León et al., 2013, 2014, 2016; Rafiq y Cacciari, 2003; Sarikaya y Bochmann, 1984; Ural y Williams, 2006; Walter et al., 1998). Another way of describing this is to assert that events x and y commute (are independent) if they occur at different ports. With such a basis, previous conformance relations for distributed testing are strongly related to the notion of a partial-commutation in which certain events commute (see, for example, Mazurkiewicz (1984)).

It seems sensible to consider that $!o_1^m!o_2$ and $!o_2!o_1^m$ are observationally equivalent for small values of m but it will cease to make sense if m becomes large enough.[1] For example, consider that each event takes at least one second to be performed and suppose that $m = 90,000$. In this case, if we observe the trace $!o_1^m!o_2$ then we have that the occurrence of $!o_2$ and the first occurrence of $!o_1$ are separated by more than one day; one might then expect to be able to distinguish between $!o_1^m!o_2$ and $!o_2!o_1^m$. We might therefore want to use a stronger notion of observational equivalence, or similarity, which conduce these traces to not being equivalent (Hierons et al., 2018). As usual, for two traces σ_1 and σ_2 to be observationally equivalent the authors require them to have the same set of projections at the ports. In addition, they also check that the *distance* between σ_1 and σ_2 is at most k, for a notion of distance. The authors define two different distances, the final results are the same but the intermediate results and proofs are different. In this thesis

[1] We are assuming, as usual, that processes are *non-Zeno*, that is, they cannot produce and infinite number of events in a finite amount of time.

we will consider that the *distance* between two traces σ_1 and σ_2, which have the same sets of projections, is the length of the shortest sequence of swaps of adjacent events, that occur at different ports, able to transform σ_1 into σ_2.

Example 1 *Consider the following two traces:* $\sigma_1 = ?i_1?i_3!o_2!o_1?i_2!o_2$ *and* $\sigma_2 = ?i_1!o_1!o_2?i_2!o_2?i_3$. *We assume that the index denotes the port at which the action is performed. We can easily check that the projections of each trace at each port is the same. These are:*

$$\begin{aligned}\pi_1(\sigma_1) = \pi_1(\sigma_2) &= ?i_1!o_1\\ \pi_2(\sigma_1) = \pi_2(\sigma_2) &= !o_2?i_2!o_2\\ \pi_3(\sigma_1) = \pi_3(\sigma_2) &= ?i_3\end{aligned}$$

The following is one of the shortest sequences of swaps that allows us to transform σ_1 *into* σ_2*:*

$$\begin{aligned}\sigma_1 = ?i_1?i_3!o_2!o_1?i_2!o_2 &\sim^1 ?i_1!o_2?i_3!o_1?i_2!o_2\\ &\sim^1 ?i_1!o_2!o_1?i_3?i_2!o_2\\ &\sim^1 ?i_1!o_2!o_1?i_2?i_3!o_2\\ &\sim^1 ?i_1!o_2!o_1?i_2!o_2?i_3\\ &\sim^1 ?i_1!o_1!o_2?i_2!o_2?i_3 = \sigma_2\end{aligned}$$

Therefore, we have $d(\sigma_1, \sigma_2) = 5$.

In this thesis, similar to the original work (Hierons et al., 2018), we formally define a conformance relation focusing on the problem of deciding whether N is a correct implementation of M and we also restrict ourselves to finite models as non-trivial decision problems with infinite models are undecidable.

1.2 Goals

The main goal of this thesis is to obtain a good understanding of the ideas behind formal testing distributed systems. Since this field is very broad, the work has concentrated on implementation relations on the distributed architecture. More precisely, we have considered recent work where a certain distance is used to complement the decision procedure concerning the possible conformance of an SUT with respect to a specification.

In addition to obtain a good knowledge of the existing theory, a second goal was to generate an original contribution. In this line, we started to work on the definition of an effective algorithm to decide the aforementioned conformance between an SUT and a specification. In order to do so, we

slightly extend the theory developed in previous work. A last goal of this thesis consists in providing a working implementation of the algorithms. Our code is self-contained, meaning that no additional libraries are needed to run it, and therefore, it is easy to use and modify to include optimizations.

Finally, we consider that this thesis is worthy of being the basis of a research paper and as such we plan to submit our main results to an appropriate conference.

1.3 Work plan

This work is based on the work by Robert M. Hierons, Mercedes G. Merayo and Manuel Núñez (Hierons et al., 2018). The last two authors are the supervisors of this thesis. In this thesis we go one step forward on the process of test derivation with respect to the previous work. We will need to extend the theoretical framework with some results that will be the basis to define and fully implement a new test derivation algorithm. Next we give more details about these two components of the thesis (theory and implementation).

1.3.1 Theoretical work plan

Our theoretical results are presented in the next chapter of this thesis. We present the background and the results that we need in order to decide whether an implementation conforms to a specification, as well as some examples to simplify its comprehension. Since this algorithm relies on novel results, we first need to extend the existing theory (Hierons et al., 2018). Chapter 2 explains the theoretical parts of this work in three steps:

- Section 2.1 defines the basic notions of the models and tools we use.
- Section 2.2 reviews the classical **dioco** conformance relation and introduces some concepts that will be used in the next section.
- Section 2.3 develops what is required for us to adequately implement the testing algorithm.

1.3.2 Implementation work plan

The implementation section consists in developing the required software to test whether an implementation conforms to a specification. This is studied in Chapter 3. This chapter is structured around two parts:

- Section 3.1 shows how the structure of `IOTSs` and finite automatas are in the software, what the format for the initial implementation and specification is, and how each of them is initialized.
- Section 3.2 presents how the program is executed, and how the models interact with each other, as well as showing a thorough example of the execution.

1.3.3 Conclusions

In addition to the previous chapters, the thesis also includes a chapter devoted to *conclusions*. In Chapter 4 we discuss some of the results and conclusions from both the theoretical and the implementation parts of the thesis.

Chapter 2

Theoretical Framework

In this chapter we give the preliminary definitions to set the testing framework and present the new theoretical results that will allow us to introduce an algorithm, based on the classical product machine construction, to decide conformance between an SUT and a specification. We will review some definitions and results from previous work Hierons et al. (2018). We also present some examples of the main notions, so it becomes easier for the reader to understand these concepts. The structure of this chapter is the following. In section 2.1 we describe the basic notions of the underlying theory. In section 2.2 we review the definition of the **dioco** relation and a distance, based on the *swaps* needed to transform one sequence into another, to bound such relation. Finally, in section 2.3 we present the theoretical framework that will justify the product machine construction that we have implemented to decide conformance between an SUT and a specification.

2.1 Preliminares

First we recall some notation concerning alphabets, sequences of actions and ports.

Given a set A, we let A^* denote the set of finite sequences of elements of A; $\epsilon \in A^*$ denotes the empty sequence. A^k denotes the set of sequences with length $k \geq 1$. Given a sequence $\sigma \in A^*$, we have that $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a.

We let I be the set of inputs and O the set of outputs. We also let $\mathcal{P}orts = \{1, \ldots, m\}$ be the set of the names of the ports and assume that the sets I and O are partitioned into sets $I_1, \ldots, I_m$ and $O_1, \ldots, O_m$ such that for all $p \in \mathcal{P}orts$, I_p and O_p are the sets of inputs and outputs at port p, respectively. We assume that $I_1, \ldots I_m, O_1, \ldots, O_m$ are pairwise disjoint. In order to distinguish between input and output we usually precede the name of an input by ? and precede the name of an output by !. If we label an action with a subindex, then the value indicates the port where the action is performed. If we label an action with a superindex, then the value indicates the position in a sequence of actions of that action.

2.1.1 Input Output Transition Systems

An Input Output Transition System (`IOTS`) is a labelled transition system in which we distinguish between input and output. We use this formalism to define processes. As usual, all the states of a specification will be final. However, we will use the notion of non-final states to denote *auxiliary* states that should not represent the end of a trace of a system. Later, we will give some examples that will help us to explain how we will take advantage of the distinction between a state that is final and one that is not for designing our algorithms.

Definition 1 *An* input output transition system *(`IOTS`) is defined by a tuple $M = (Q, Q_F, I, O, T, q_{in})$ in which Q is a countable set of states, $Q_F \subseteq Q$ is the set of final states, $q_{in} \in Q$ is the initial state, I is a countable set of inputs, O is a countable set of outputs, and $T \subseteq Q \times (I \cup O) \times Q$ is the transition relation. A transition $(q, a, q') \in T$ means that from state q it is possible to move to state q' with action $a \in I \cup O$.*

We say that a state $q \in Q$ is quiescent *if from q it is not possible to take a transition whose action is an output without first receiving an input. We extend T to T_δ by adding transition (q, δ, q) for each quiescent state q. We say that M is* input-enabled *if for all $q \in Q$ and $?i \in I$ there is some $q' \in Q$ such that $(q, ?i, q') \in T$.*

We let $\mathcal{A}ct$ denote the set of actions, that is, $\mathcal{A}ct = I \cup O \cup \{\delta\}$. Given port $p \in \mathcal{P}orts$, $\mathcal{A}ct_p = I_p \cup O_p \cup \{\delta\}$ denotes the set of observations that can be made at p. We define the function port $: I \cup O \longrightarrow \mathcal{P}orts$ such that $port(a) = p$ if $a \in \mathcal{A}ct_p$. Abusing the notation, we will assume that $port(\delta) = p$ holds for all $p \in \mathcal{P}orts$.

We let `IOTS`$(I, O, \mathcal{P}orts)$ denote the set of `IOTSs` with input set I, output set O and port set $\mathcal{P}orts$.

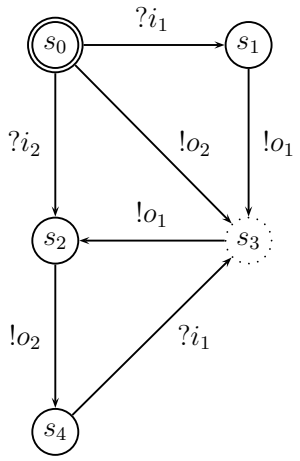


Figure 2.1: A diagrammatic representation of an IOTS

As a process can be identified with its initial state and we can define a process corresponding to a state q of M by making q the initial state, we will use states and processes and their notation interchangeably. By default, all the states of an IOTS are final (as we previously said, non-final states will be included for the construction of some auxiliary processes).

An IOTS can be represented by a diagram. Figure 2.1 shows an example of the graphical representation of an IOTS. Nodes represent states of the IOTS and transitions are represented by arcs between the nodes. We use a double circle to denote the initial state, s_0, while dotted circles denote non-final states and non-dotted circles denote final states. In this case, all the states are final except s_3. The state s_0 could receive either the input $?i_1$, at port 1, or $?i_2$, at port 2 and will move to states s_1 and s_2, respectively. In addition, the initial state s_0 could produce the output $!o_2$, at port 2, reaching state s_3 from which only the output transition $!o_1$ could be produced. We can also see that this IOTS is not input-enabled. For example, there are no outgoing transitions from s_1 labelled by either $?i_1$ or $?i_2$.

A *trace* is a sequence of observable actions that takes a process from the initial state to a final state. The set of traces of an IOTS constitutes the language of the process.

Definition 2 *Let $M = (Q, Q_F, I, O, T, q_{in})$ be an IOTS. We use the following notation.*

1. *If $(q, a, q') \in T_\delta$, for $a \in \mathcal{A}ct$, then we write $q \xrightarrow{a} q'$.*

2. *Let $\sigma = a^1 \dots a^k \in \mathcal{A}ct^*$ be a finite sequence of actions. We write*

$q \stackrel{\sigma}{\Longrightarrow} q'$ *if there exist* $q_0, \ldots, q_k \in Q$ *such that* $q = q_0$, $q' = q_k$ *and for all* $1 \leq i \leq k$ *we have that* $q_{i-1} \xrightarrow{a^i} q_i$.

3. *We write* $q \stackrel{\sigma}{\Longrightarrow}$ *if there exists* $q' \in Q$ *such that* $q \stackrel{\sigma}{\Longrightarrow} q'$.

4. *We write* $M \stackrel{\sigma}{\Longrightarrow}$ *if* $q_{in} \stackrel{\sigma}{\Longrightarrow}$.

5. *Let* $\rho = (q_0, a^1, q_1)(q_1, a^2, q_2)\ldots(q_{k-1}, a^k, q_k)$ *be a sequence of consecutive transitions. We say that the label of* ρ, *denoted by* $label(\rho)$, *is the sequence of actions associated with the transitions, that is,* $label(\rho) = a^1 \ldots a^k$.

We define the language of M *as* $L(M) = \{\sigma | q_{in} \stackrel{\sigma}{\Longrightarrow} q \wedge q \in Q_F\}$. *We say that* $\sigma \in L(M)$ *is a* trace *of* M. *We let* $L_\delta(M) = \{\sigma | q_{in} \stackrel{\sigma\delta}{\Longrightarrow} q \wedge q \in Q_F\}$; *the set of traces of* M *that can take* M *to a quiescent state.*

Note that for every state q we have $q \stackrel{\epsilon}{\Longrightarrow} q$ holds. Therefore, $\epsilon \in L(M)$ for every process M.

2.1.2 Finite automata

We will consider finite automata because our proposal will rely in the product of two automata. It could be thought that finite automata are exactly the same as `IOTSs` but there is a notable difference that we will exploit in Section 3.1. Specifically, there is no constraints about the use of δ transitions in finite automata while an `IOTS` only can have a δ transition outgoing from a state if there are no possible outputs from such state. In addition, there exist more tools to deal with finite automata, in particular for determinizing a system, and generating product machines, which we will apply.

Definition 3 *A* finite automata *is defined by a tuple* $M = (Q, Q_F, \mathcal{A}ct, T, q_{in})$ *in which* Q *is a finite set of states,* $Q_F \subseteq Q$ *is the set of final states,* $q_{in} \in Q$ *is the initial state,* $\mathcal{A}ct$ *is a finite set of actions, and* $T \subseteq Q \times \mathcal{A}ct \times Q$ *is the transition relation. A transition* $(q, a, q') \in T$, *also denoted by* $q \xrightarrow{a} q'$, *means that from state* q *it is possible to move to state* q' *with action* a.

We say that M *is* deterministic *if for all* $q \in Q$ *and* $a \in \mathcal{A}ct$ *there exists at most one state* $q' \in Q$ *such that* $(q, a, q') \in T$. *Otherwise, we say that* M *is* non-deterministic.

In this thesis, since finite automata will be constructed from `IOTSs`, we will usually have that the set of actions $\mathcal{A}ct$ is defined as $\mathcal{A}ct = I \cup O \cup \{\delta\}$. The next definition is a trivial adaption to the finite automata framework of Definition 2.

Definition 4 *Let $M = (Q, Q_F, \mathcal{A}ct, T, q_{in})$ be a finite automata. We use the following notation.*

1. *If $(q, a, q') \in T_\delta$, for $a \in \mathcal{A}ct$, then we write $q \xrightarrow{a} q'$.*
2. *Let $\sigma = a^1 \dots a^k \in \mathcal{A}ct^*$ be a finite sequence of actions. We write $q \stackrel{\sigma}{\Longrightarrow} q'$ if there exist $q_0, \dots, q_k \in Q$ such that $q = q_0$, $q' = q_k$ and for all $1 \leq i \leq k$ we have that $q_{i-1} \xrightarrow{a^i} q_i$.*
3. *We write $q \stackrel{\sigma}{\Longrightarrow}$ if there exists $q' \in Q$ such that $q \stackrel{\sigma}{\Longrightarrow} q'$.*
4. *We write $M \stackrel{\sigma}{\Longrightarrow}$ if $q_{in} \stackrel{\sigma}{\Longrightarrow}$.*
5. *Let $\rho = (q_0, a^1, q_1)(q_1, a^2, q_2) \dots (q_{k-1}, a^k, q_k)$ be a sequence of consecutive transitions. We say that the label of ρ, denoted by $label(\rho)$, is the sequence of actions associated with the transitions, that is, $label(\rho) = a^1 \dots a^k$.*

We define the language of M as $L(M) = \{\sigma | q_{in} \stackrel{\sigma}{\Longrightarrow} q \wedge q \in Q_F\}$. We say that $\sigma \in L(M)$ is a trace *of M. We let $L_\delta(M) = \{\sigma | q_{in} \stackrel{\sigma\delta}{\Longrightarrow} q \wedge q \in Q_F\}$.*

2.2 The bounded dioco conformance relation

In this section we review some specific notions concerning the **dioco** relation (Hierons et al., 2008, 2012) When comparing two `IOTSs` we will assume that they have the same set of ports and the same set of actions $\mathcal{A}ct_p$ for all $p \in \mathcal{P}orts$. Moreover, as usual, we require that SUTs are *input-enabled*. Note that this is not a strong restriction because we can always assume that if an input cannot be applied in some state of the SUT, then we can consider that there is a response to the input that reports that this input is blocked. We also assume that specifications are input-enabled since this assumption simplifies the analysis. However, as it was already shown (Hierons, 2016; Hierons et al., 2012), it is possible to remove this restriction in our framework. As usual, for work related to distributed testing from `IOTSs`, we require that processes are not output-divergent: there cannot be a state from which there is an infinite sequence of consecutive transitions whose labels are all outputs. Conformance relations for distributed testing are usually based on an equivalence relation $\sim$ on traces. Essentially, the relation $\sim$ shows the fact that in distributed testing each tester observes only the events at its port and this matches to a projection of the global trace that occurred. Therefore, traces that cannot be differentiated must be considered to be equivalent, that is, we should not be able to distinguish between them.

Definition 5 *Let $p \in \mathcal{P}orts$ and $\sigma \in \mathcal{A}ct^*$ be a sequence of actions. We let $\pi_p(\sigma)$ denote the projection of σ onto port p and $\pi_p(\sigma)$ is called a* local trace. *Formally,*

$$\pi_p(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ a\pi_p(\sigma') & \text{if } \sigma = a\sigma' \wedge a \in \mathcal{A}ct_p \\ \pi_p(\sigma') & \text{if } \sigma = a\sigma' \wedge a \in \mathcal{A}ct \setminus \mathcal{A}ct_p \end{cases}$$

Given $\sigma, \sigma' \in \mathcal{A}ct^*$ we write $\sigma \sim \sigma'$ if σ and σ' cannot be distinguished when making local observations, that is, for all $p \in \mathcal{P}orts$ we have that $\pi_p(\sigma) = \pi_p(\sigma')$. Clearly, $\sim$ is an equivalence relation and we will denote by $[\sigma]$ the equivalence class of σ, that is, $[\sigma] = \{\sigma' \in \mathcal{A}ct^* | \sigma \sim \sigma'\}$.

In this thesis we consider a conformance relation based on **dioco** in which an SUT should not show behaviours that do not appear in the specification, except for the order between events performed at different ports (the interested reader is referred to the original work (Hierons et al., 2014, 2018) for more details).

Definition 6 *Let $M, N \in \mathtt{IOTS}(I, O, \mathcal{P}orts)$. We write N* **dioco** *M if and only if for every quiescent trace $\sigma\delta \in L(N)$, there exists a trace $\sigma' \in L(M)$ such that $\sigma' \sim \sigma\delta$.*

Next, we introduce a notion of distance and the conformance relation that is based on this metric. The idea is that we will not consider two traces to be *equivalent* if the distance between these traces is bigger than a certain threshold. In this thesis we will consider that the distance between traces is given by the minimum number of *swaps* that we need to transform one trace into the other. Actually, since it is possible to use local projections to distinguish between traces that are not related under $\sim$, it will only be necessary to define the distance between traces σ_1 and σ_2 if $\sigma_1 \sim \sigma_2$. Finally, note that in Definition 5 we implicitly use that $\delta \in \mathcal{A}ct_p$ for all port $p \in \mathcal{P}orts$. This fact will be relevant when we define *valid* swaps because swaps must involve actions in two different ports and δ belongs to all the ports.

Definition 7 *Let $\sigma, \sigma' \in \mathcal{A}ct^*$ be sequences of actions. We write $\sigma \sim^1 \sigma'$ if there exist $p, q \in \mathcal{P}orts$, with $p \neq q$, $\sigma_1, \sigma_2 \in \mathcal{A}ct^*$, $a \in \mathcal{A}ct_p \setminus \{\delta\}$ and $b \in \mathcal{A}ct_q \setminus \{\delta\}$ such that $\sigma = \sigma_1 ab \sigma_2$ and $\sigma' = \sigma_1 ba \sigma_2$.*

Let $\sigma, \sigma' \in \mathcal{A}ct^*$ be sequences of actions such that $\sigma \sim \sigma'$. We define the distance between these two traces, denoted by $d(\sigma, \sigma')$, to be the smallest integer k such that there exist $\sigma_0, \sigma_1, \ldots, \sigma_k$ where $\sigma = \sigma_0$, $\sigma' = \sigma_k$, and for all $0 \leq i < k$ we have that $\sigma_i \sim^1 \sigma_{i+1}$.

Next we present some examples to illustrate how allowed swaps can be done. We have already seen in Example 1 how swaps can be produced. We will see another sequence of swaps for this example. Actually, it is important to emphasize that, in general, there is not a unique sequence of swaps to transform a trace into another.

Example 2 *Consider the following two traces:* $\sigma_1 = ?i_1?i_3!o_2!o_1?i_2!o_2$ *and* $\sigma_2 = ?i_1!o_1!o_2?i_2!o_2?i_3$. *We used them already in Example 1. We already know that the projections of each trace at each port is the same and we have shown how* σ_1 *can be transformed into* σ_2. *Another of the shortest sequences of swaps that allows us to transform* σ_1 *into* σ_2 *is:*

$$\begin{aligned}\sigma_1 = ?i_1?i_3!o_2!o_1?i_2!o_2 &\sim^1 ?i_1?i_3!o_1!o_2?i_2!o_2\\ &\sim^1 ?i_1!o_1?i_3!o_2?i_2!o_2\\ &\sim^1 ?i_1!o_1!o_2?i_3?i_2!o_2\\ &\sim^1 ?i_1!o_1!o_2?i_2?i_3!o_2\\ &\sim^1 ?i_1!o_1!o_2?i_2!o_2?i_3 = \sigma_2\end{aligned}$$

Of course, the distance is still the same, that is, $d(\sigma_1, \sigma_2) = 5$.

The next example includes a δ transition in the traces.

Example 3 *Consider now the following traces* $\sigma_1 = ?i_1!o_2\delta?i_2!o_1?i_1!o_2$ *and* $\sigma_2 = !o_2?i_1\delta!o_1?i_1?i_2!o_2$. *As usual, the index denotes the port at which the action is performed. By checking the projections of each trace at each port, we obtain* $\pi_1(\sigma_1) = \pi_1(\sigma_2) = ?i_1\delta!o_1?i_1$ *and* $\pi_2(\sigma_1) = \pi_2(\sigma_2) = !o_2\delta?i_2!o_2$. *Now we show one of the shortest sequences of swaps to transform* σ_1 *into* σ_2*:*

$$\begin{aligned}\sigma_1 = ?i_1!o_2\delta?i_2!o_1?i_1!o_2 &\sim^1 !o_2?i_1\delta?i_2!o_1?i_1!o_2\\ &\sim^1 !o_2?i_1\delta!o_1?i_2?i_1!o_2\\ &\sim^1 !o_2?i_1\delta!o_1?i_1?i_2!o_2 = \sigma_2\end{aligned}$$

We have $d(\sigma_1, \sigma_2) = 3$.

Finally, the next example shows why it is important to carefully deal with δ transitions.

Example 4 *Consider the traces* $\sigma_1 = ?i_1!o_1\delta?i_2!o_2$ *and* $\sigma_2 = ?i_1?i_2\delta!o_1!o_2$. *Checking projections, we obtain* $\pi_1(\sigma_1) = ?i_1!o_1\delta$ *while* $\pi_1(\sigma_2) = ?i_1\delta!o_1$ *and* $\pi_2(\sigma_1) = \delta?i_2!o_2$ *while* $\pi_2(\sigma_2) = ?i_2\delta!o_2$. *Once it is clear that the projections are not the same, we can conclude that there is no sequence of allowed swaps able to transform* σ_1 *into* σ_2. *If we are not careful, we might swap* δ *and* $?i_2$ *in* σ_1 *and, after another swap, we would obtain* σ_2.

Given a trace σ and $k \geq 0$, we define the language of traces in $[\sigma]$ whose distance from σ is at most k.

Definition 8 *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions and $k \geq 0$. We say that $L_k(\sigma)$ is equal to $\{\sigma' \in [\sigma] | d(\sigma', \sigma) \leq k\}$.*

When looking at the distance d, we can define what it means for a trace to be allowed if the specification is M and we allow at most distance k.

Definition 9 *Let $M \in \mathtt{IOTS}(I, O, \mathcal{P}orts)$, $k \geq 0$, and $\sigma \in \mathcal{A}ct^*$. We say that σ is allowed by M given k if there is some trace $\sigma' \in L(M)$ such that $\sigma' \sim \sigma$ and $d(\sigma, \sigma') \leq k$.*

Next, we introduce the conformance relation that considers the distance as a restriction for distributed testing.

Definition 10 *Let $M, N \in \mathtt{IOTS}(I, O, \mathcal{P}orts)$ be two systems and $k \geq 0$. We write N $dioco^k_{sw}$ M if every trace $\sigma \in L_\delta(N)$ is allowed by M given d and k.*

Given an IOTS M we can define the set of sequences that are within a certain distance of traces of M.

Definition 11 *Let $M \in \mathtt{IOTS}(I, O, \mathcal{P}orts)$ and $k \geq 0$. We define the set $L_k(M) \subseteq \mathcal{A}ct^*$ as*

$$L_k(M) = \{\sigma | \exists \sigma' \in L(M) : \sigma' \sim \sigma \wedge d(\sigma, \sigma') \leq k\}$$

We have that $L_k(M)$ denotes the set of traces that are at distance at most k from traces of M.

The following proposition is straightforward taking into account that a sequence σ is allowed by M given d and k if and only if there is some trace $\sigma' \in L(M)$ with $d(\sigma', \sigma) \leq k$. Note that we need to assume that M and N are input-enabled and are not output-divergent.

Proposition 1 *Let $M, N \in \mathtt{IOTS}(I, O, \mathcal{P}orts)$ be two systems and $k \geq 0$. We have N $\mathbf{dioco^k_{sw}}$ M if and only if $L_\delta(N) \subseteq L_k(M)$.*

2.3 Establishing conformance

In this section we present our proposal for determining the conformance of an SUT with respect to a specification when we consider the conformance relation $\mathbf{dioco^k_{sw}}$.

$$
\begin{aligned}
Q' = \; & \{q^1 | q \in Q\} \cup \{q^2 | q \in Q\} \cup \\
& \{q_{t_1 t_2} | \exists t_1 = (q_1, a, q_2) \in T, t_2 = (q_2, b, q_3) \in T : port(a) \neq port(b)\} \\
Q'_F = \; & \{q^1 | q \in Q_F\} \cup \{q^2 | q \in Q_F\}
\end{aligned}
$$

$$
\begin{aligned}
T' = \; & \{(q_1^1, a, q_2^1) | (q_1, a, q_2) \in T\} \cup \{(q_1^2, a, q_2^2) | (q_1, a, q_2) \in T\} \cup \\
& \left\{ (q_1^1, b, q_{t_1 t_2}), (q_{t_1 t_2}, a, q_3^2) \;\middle|\; \begin{array}{l} \exists q_2 \in Q, t_1 = (q_1, a, q_2) \in T, \\ t_2 = (q_2, b, q_3) \in T : port(a) \neq port(b) \end{array} \right\}
\end{aligned}
$$

Figure 2.2: Description of Q', Q'_F and T' in Definition 12

Based on Proposition 1, we will put our effort on simplifying the way of showing whether $L_\delta(N) \subseteq L_k(M)$. To do it, we will use the results that are introduced in (Hierons et al., 2014, 2018). They establish that the sets $L_k(M)$ are regular. First, a finite automaton, $\mathcal{M}(M, k)$, that recognises $L_k(M)$ is defined. Second, an algorithm that implements the construction of this automaton is proposed.

Definition 12 *Let $M \in$ IOTS$(I, O, \mathcal{P}orts)$ and $k \geq 0$. We inductively define the IOTS $\mathcal{M}(M, k)$ as follows. If $k = 0$ then $\mathcal{M}(M, k) = M$. Otherwise, if $k > 0$ then let us suppose that $\mathcal{M}(M, k-1) = (Q, Q_F, I, O, T, q_{in})$. Then $\mathcal{M}(M, k) = (Q', Q'_F, I, O, T', q'_{in})$ in which $q'_{in} = q^1_{in}$ and Q', Q'_F and T' are given in Figure 2.2.*

In Algorithm 1 we show the pseudocode that generates $\mathcal{M}(M, k)$. Essentially, the construction of the $\mathcal{M}(M, k)$ proceeds through k steps. Initially, all the states and transitions of M are duplicated. Then, for each pair of consecutive transitions with events at different ports, an auxiliary (non final) state and two new transitions are included to capture the swap of these events. This process is applied to the new machine k times.

Example 5 *Consider the IOTS M depicted in Figure 2.3 (a) in which indexes denote the port where the actions are produced. Figure 2.3 (b) shows $\mathcal{M}(M, 1)$, where we have considered two pairs of consecutive transitions:*

- $t = (q_0, a_1, q_1)$ *and* $t' = (q_1, b_2, q_2)$.
- $t' = (q_1, b_2, q_2)$ *and* $t'' = (q_2, c_3, q_3)$.

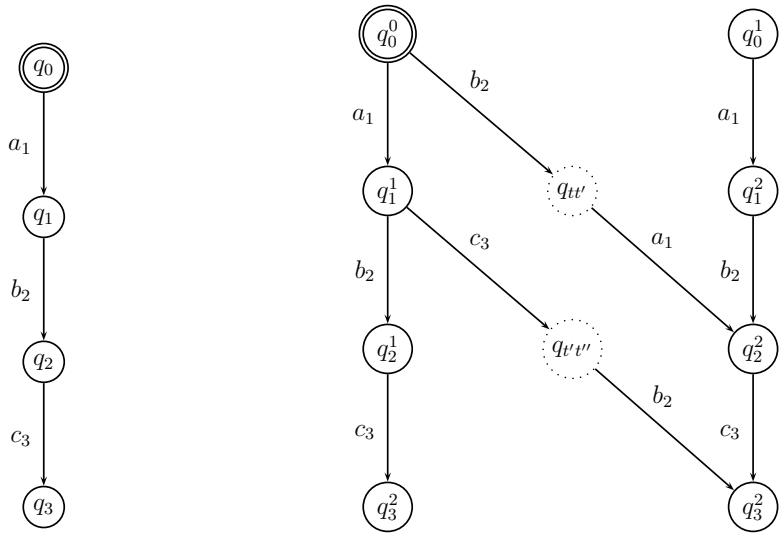
Finally, the following result states that the language accepted by the automaton constructed by using Algorithm 1 coincides with the one given by Definition 11.

Algorithm `Produce_`$\mathcal{M}(M,k)$
/*$M = (Q, Q_F, I, O, T, q_{in})$, $M' = (Q', Q'_F, I, O, T', q'_{in})$*/
if $k = 1$ **then**
  $Q' := \emptyset; Q'_F := \emptyset; T' := \emptyset;$
  foreach *state* $q \in Q$ **do**
    $Q' := Q' \cup \{q^1, q^2\};$
    /*q^1, q^2 are fresh states*/
    if $q \in Q_F$ **then** $Q'_F := Q'_F \cup \{q^1, q^2\};$
  end
  $q'_{in} := q^1_{in};$
  foreach *transition* $(q_1, a, q_2) \in T$ **do**
    $T' := T' \cup \{(q^1_1, a, q^1_2), (q^2_1, a, q^2_2)\}$
  end
  foreach *pair of transitions* $t = (q_1, a, q_2), t' = (q_2, b, q_3) \in T$ *with* $port(a) \neq port(b)$ **do**
    $Q' := Q' \cup \{q_{tt'}\};$
    /*$q_{tt'}$ is a fresh state*/
    $T' := T' \cup \{(q^1_1, b, q_{tt'}), (q_{tt'}, a, q^2_3)\};$
  end
  return$(M');$
else
  return(`Produce_`$\mathcal{M}$(`Produce_`$\mathcal{M}(M, k-1), 1)$);
end

Algorithm 1: Producing $\mathcal{M}(M,k)$ for $k > 0$.

Theorem 1 *Let* $M \in$ `IOTS`$(I, O, \mathcal{P}orts)$ *and* $k \geq 0$. *We have that* $L_k(M) = L(\mathcal{M}(M,k))$.

In order to determine the conformance of an SUT N with respect to a specification M under the $dioco^k_{sw}$ conformance relation we must prove that $L_\delta(N) \subseteq L_k(M)$ as Proposition 1 establishes. For doing it, we propose the use of a *product machine*. A product machine is classically used to decide whether two finite automata accept either the same language or one accepts a subset of the other. Essentially, it simulates the parallel execution of the two finite automata as long as the actions agree; if the one playing the role of SUT shows a mismatch then a failure is detected and the product machine leads to a fail state. Therefore, the SUT does not conform to a specification if the fail state of the product machine is reachable from its initial state. Next we formally define how the product machine is constructed.



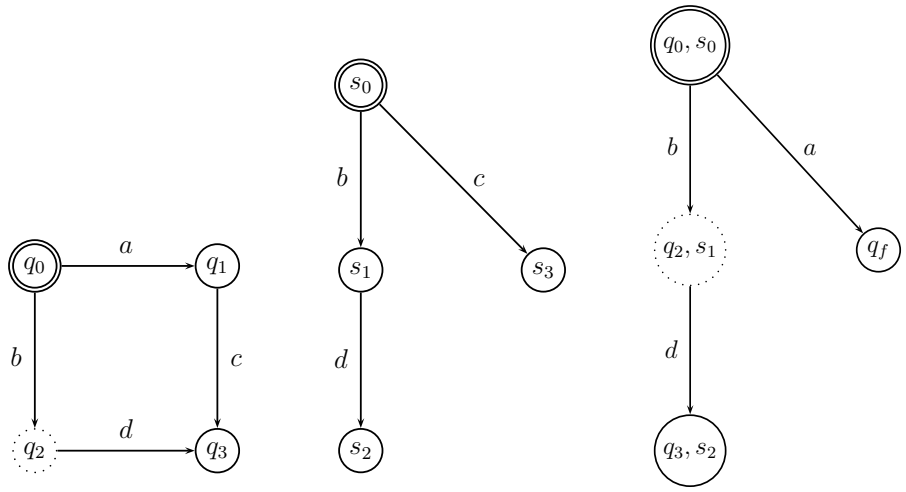
(a) M (b) $\mathcal{M}(M, 1)$

Figure 2.3: Generation of $\mathcal{M}(M, 1)$ by Algorithm 1

Definition 13 *Let* $M_1 = (Q^1, Q_F^1, \mathcal{A}ct, T^1, q_{in}^1)$ *and* $M_2 = (Q^2, Q_F^2, \mathcal{A}ct, T^2, q_{in}^2)$ *be two* deterministic *finite automata. The* product machine *of* M_1 *and* M_2*, denoted by* $P(M_1, M_2)$*, is the tuple* $((Q^1 \times Q^2) \cup \{q_f\}, Q_F^1 \times Q_F^2 \cup \{q_f\}, \mathcal{A}ct, T, (q_{in}^1, q_{in}^2))$ *where* q_f *is a fresh state and* T *is the transition relation defined as follows:*

- *For all* $(q_1, q_2) \in Q^1 \times Q^2$ *and for all* $a \in \mathcal{A}ct$ *such that there exist* $(q_1, a, q_1') \in T^1$ *and* $(q_2, a, q_2') \in T^2$ *we have that* $((q_1, q_2), a, (q_1', q_2')) \in T$*.*
- *For all* $(q_1, q_2) \in Q^1 \times Q^2$ *and for all* $a \in \mathcal{A}ct$ *such that there exists* $(q_1, a, q_1') \in T^1$ *and there does not exist* $(q_2, a, q_2') \in T^2$ *we have that* $((q_1, q_2), a, q_f) \in T$*.*

Figure 2.4 depicts automata M_1 and M_2 and the product machine corresponding to these finite automata. For the sake of clarity, we have omitted the states that are not reachable. Both automata, M_1 and M_2, present a transition outgoing from the initial state that is labelled with b. According to the definition of $P(M_1, M_2)$, the machine has a transition from (q_0, s_0) leading to the state (q_2, s_1) and labelled with b. However, while the automaton M_1 presents a transition from the initial state labelled with a, there does not exist a corresponding transition in the automaton M_2. Therefore, the product machine includes a transition leading to the fail state q_f.



(a) M_1 (b) M_2 (c) $P(M_1, M_2)$

Figure 2.4: Automata M_1, M_2 and $P(M_1, M_2)$

We need to work with deterministic finite automata for the generation of the product machine. The non-determinism could lead the product machine to the fail state although the SUT conforms to the specification. In Figure 2.5 we show an example in which we illustrate the problem that arises when we deal with non deterministic automata. Although it is clear that ND conforms to itself, the product machine $P(ND, ND)$ leads to the fail state, as we can see in Figure 2.5 (c). Therefore, we will transform both automata N and $\mathcal{M}(M, k)$ into equivalent deterministic finite automata using the classical method (Hopcroft et al., 2006).

Although we have to prove that $L_\delta(N) \subseteq L_k(M)$, we will prove a slightly different result based on prefixes.

Definition 14 *Let $L \subseteq \mathcal{A}ct^*$ be a context-free language. We define the language that recognizes all the prefixes of L as:*

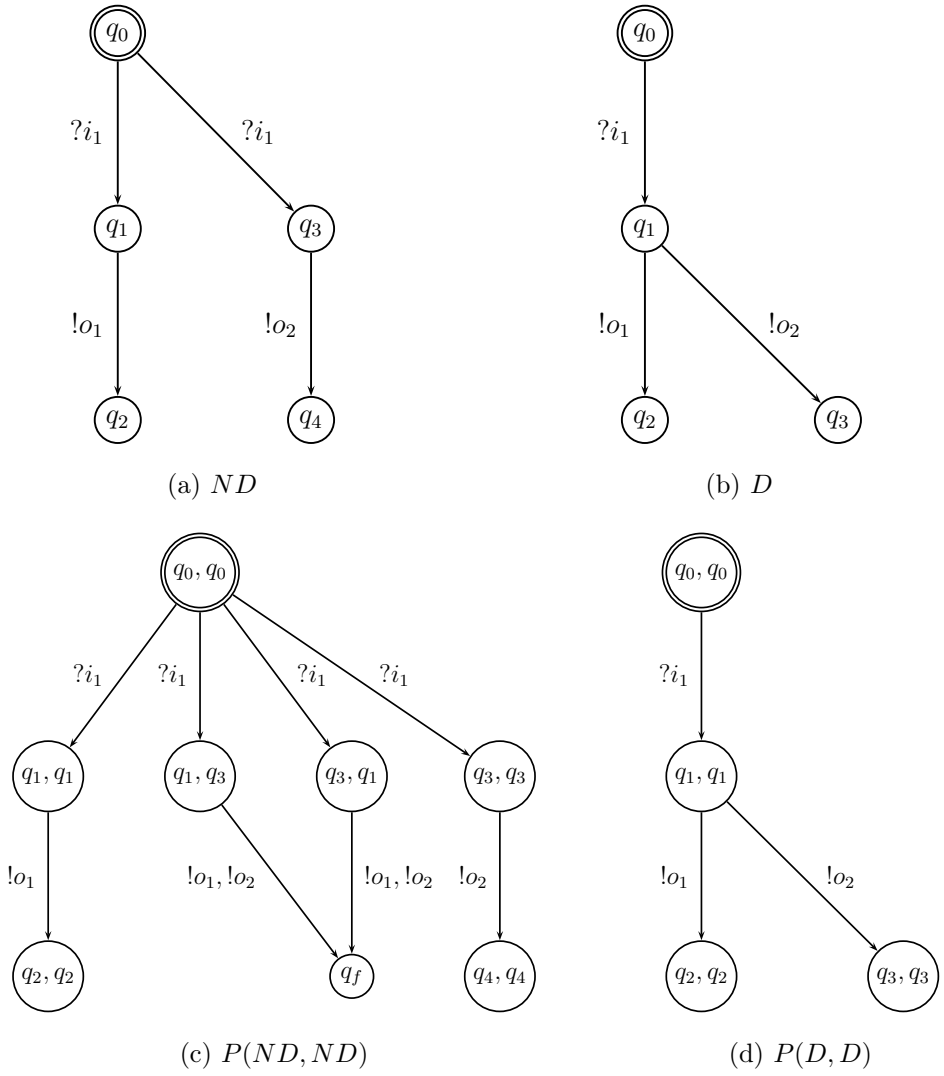
$$pref(L) = \{\sigma | \exists \sigma' \in \mathcal{A}ct^* : \sigma\sigma' \in L\}$$

The next result proves that given a trace recognized by $L_k(M)$, every quiescent prefix of that trace is also recognized by $L_k(M)$.

Lemma 1 *Let $M \in \texttt{IOTS}(I, O, \mathcal{P}orts)$ and $k \geq 0$. If $\sigma\delta\sigma' \in L_k(M)$ then $\sigma\delta \in L_k(M)$.*

Proof

By definition of $L_k(M)$, if $\sigma\delta\sigma' \in L_k(M)$ then there exist $\sigma'', \sigma''' \in \mathcal{A}ct$ such



(a) ND (b) D

(c) $P(ND, ND)$ (d) $P(D, D)$

Figure 2.5: Non-determinism and product machines

that $\sigma \sim \sigma''$, $\sigma' \sim \sigma'''$, $d(\sigma, \sigma'') + d(\sigma', \sigma''') \leq k$ *and* $\sigma''\delta\sigma''' \in L(M)$. *The previous claim holds because* δ *cannot be swapped and, therefore, the number of swaps needed to transform the original sequences is equal to the addition of the number of swaps needed to transform each of the subsequences. Now, taking into account that every state of* M *is final, we have that* $\sigma''\delta \in L(M)$. *In addition,* $0 \leq d(\sigma, \sigma'') \leq k$. *Therefore* $\sigma\delta \in L_k(M)$ *as required.* □

Proposition 2 *Let* $M, N \in \mathtt{IOTS}(I, O, \mathcal{P}orts)$ *and* $k \geq 0$. *We have that* $L_\delta(N) \subseteq L_k(M)$ *if and only if* $pref(L_\delta(N)) \subseteq pref(L_k(M))$.

Proof

We first prove the left to right implication. We assume $L_\delta(N) \subseteq L_k(M)$. *Let* $\sigma \in pref(L_\delta(N))$. *Then, there exists* σ' *such that* $\sigma\sigma' \in L_\delta(N)$ *and, by hypothesis,* $\sigma\sigma' \in L_k(M)$. *As a result,* $\sigma \in pref(L_k(M))$ *as required.*

We now prove the right to left implication. We use a proof by contraposition, that is, we assume that $L_\delta(N) \not\subseteq L_k(M)$ *and prove* $pref(L_\delta(N)) \not\subseteq pref(L_k(M))$. *If* $L_\delta(N) \not\subseteq L_k(M)$ *then there exists a sequence* $\sigma\delta \in L_\delta(N)$ *such that* $\sigma\delta \notin L_k(M)$. *If* $\sigma\delta \in L_\delta(N)$ *then, obviously,* $\sigma\delta \in pref(L_\delta(N))$. *We will prove, by contradiction, that this sequence does not belong to the set* $pref(L_k(M))$. *Let us suppose that* $\sigma\delta \in pref(L_k(M))$. *Therefore, by definition of the set of prefixes of a language, there exists* σ' *such that* $\sigma\delta\sigma' \in L_k(M)$. *Lemma 1 proves that all quiescent prefixes of* $L_k(M)$ *also belong to the language. Since* $\sigma\delta\sigma'$ *belongs to* $L_k(M)$ *and* $\sigma\delta$ *is a quiescent prefix of this trace we also have* $\sigma\delta \in L_k(M)$ *and this is a contradiction because in the beginning of the proof we assumed that* $\sigma\delta \notin L_k(M)$. □

We would like to finish this section by emphasizing the importance of the previous result to ensure the correctness of our approach. In principle, in order to show that an SUT N is faulty, thanks to Proposition 1, we need to find a trace belonging to $L_\delta(N)$ such that it does not belong to $L_k(M)$. Our product machine is constructed in such a way that any sequence of actions that N can perform but such that $L_k(M)$ cannot perform will lead to the fail state. However, this does not necessary shows that N is faulty: this sequence might not be quiescent. The previous result proves that such a non-quiescent sequence can be extended to provide a quiescent sequence belonging to N such that it does not belong to $L_k(M)$.

Chapter 3

The tool

In this chapter we briefly explain how we have implemented the software that allow us to decide whether an SUT conforms to a specification with respect to the $\mathbf{dioco}^{k}_{\mathbf{sw}}$ conformance relation. We describe technical details, specifically, the internal structure of the data, how these elements interact, and how we have implemented our theoretical proposal.

3.1 Data Structure

The implementation of the algorithm requires mainly two Java classes. Initially, we use the *IOTS* class to represent both the SUT and the specification. Later, once we have parsed the information and extended the specification model to capture the possible swaps that can be accepted according to Algorithm 1, we use the *FDA* class to represent the corresponding deterministic finite automata.

Figure 3.1 depicts the class diagram. We can see that both classes are very similar but slight differences can be seen in the structure of transitions. In both cases, the set of transitions are represented by a *Map* structure. The key of the Map is composed of a state and an action. Regarding the values associated with each of them, we distinguish between the *IOTS* and the *FDA* classes. In the case of the *IOTS* class, due to the non-deterministic nature of the formalism, we use a list of values to represent all the states reached by the transitions outgoing from each state of the model with a specific action. However, in the *FDA* class only a value is associated to a state and an action.

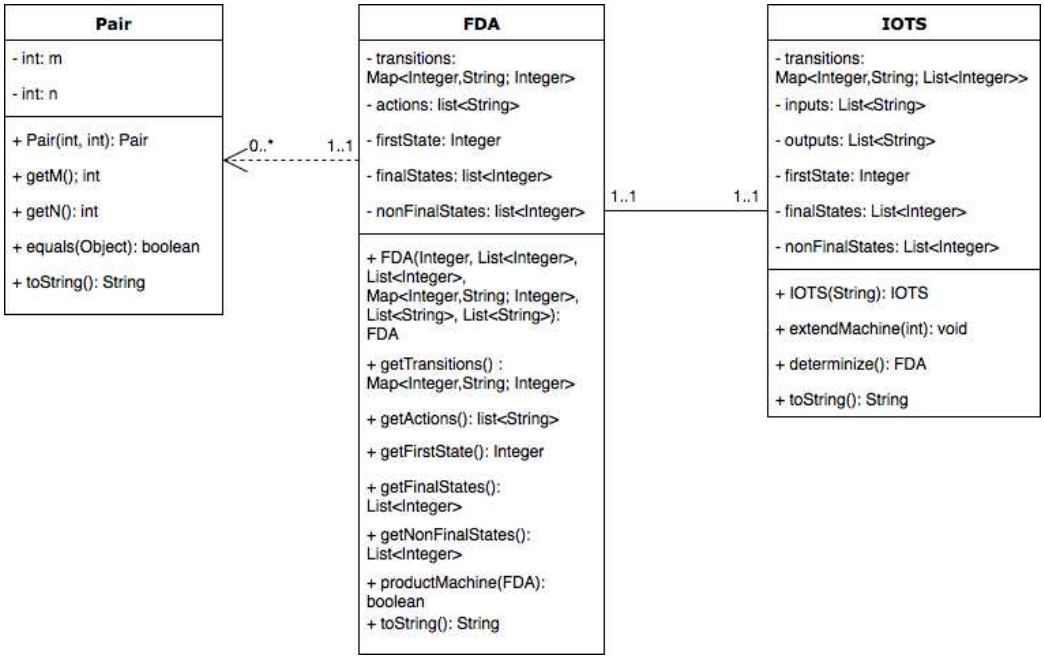


Figure 3.1: Class diagram

Another difference is related to the actions. The *IOTS* class distinguishes between inputs and output actions but in the *FDA* class this distinction disappears, we only have actions.

We consider an additional class, *Pair*, to represent states of the product machine that we will produce. This class is also used to store, in two lists, both the states that have been analyzed during the generation of the product machine and the ones that are pending.

3.2 Implementation

With the goal of deciding the conformance under the $\mathbf{dioco}^k_{\mathbf{sw}}$ implementation relation, for a given k, we have implemented our theoretical approach following the next steps that are represented in Figure 3.2.

1. The first process parses the two files that describe the specification M and the SUT N to create the corresponding `IOTSs`.

2. Once we generate both models, we construct the machine $\mathcal{M}(M, k)$ applying the Algorithm 1. This machine recognizes the set of traces that are at distance at most k from traces of M. Taking into account

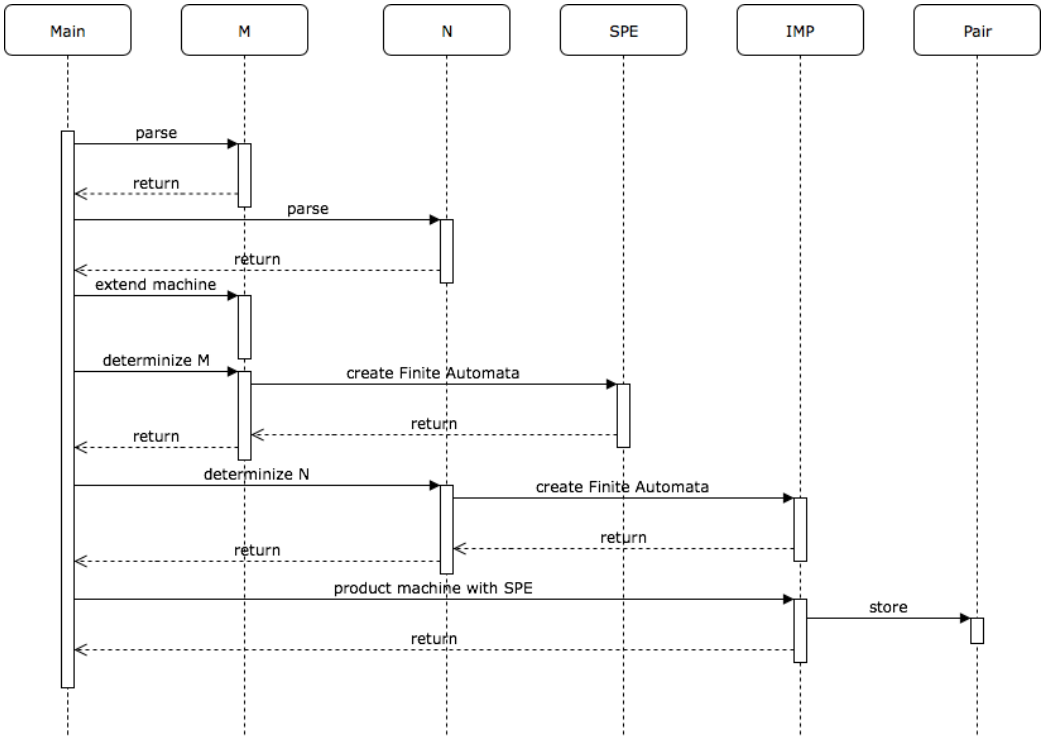


Figure 3.2: Sequence diagram

that this machine is an extension of the specification model, we do not create a new `IOTS` for $\mathcal{M}(M, k)$, we only extend M.

3. At this step we transform both N and $\mathcal{M}(M, k)$ into finite automata. Then, we will determinize both automata before producing the product machine. They are identified in Figure 3.2 as IMP and SPE, respectively.

4. Finally, we generate the product machine from N and $\mathcal{M}(M, k)$. We aim at determining if the product machine can reach the fail state. The process for creating the product machine follows a lazy evaluation. Beginning at the initial state of the product machine, we include in the model all the valid transitions outgoing from it and store those states reached by these transitions. This process allows us to create and analyze, during the construction of the product machine, only reachable states. As a consequence, as soon as we have a transition leading to a fail state, we stop the generation of the product machine and emit a verdict of non conformance of N with respect to M. In the case that the process does not produce any transition leading to the fail state,

```
0 ?i_3 0
0 !o_1 1
1 ?i_3 1
1 !o_2 2
2 ?i_3 2
2 delta 2
@
```

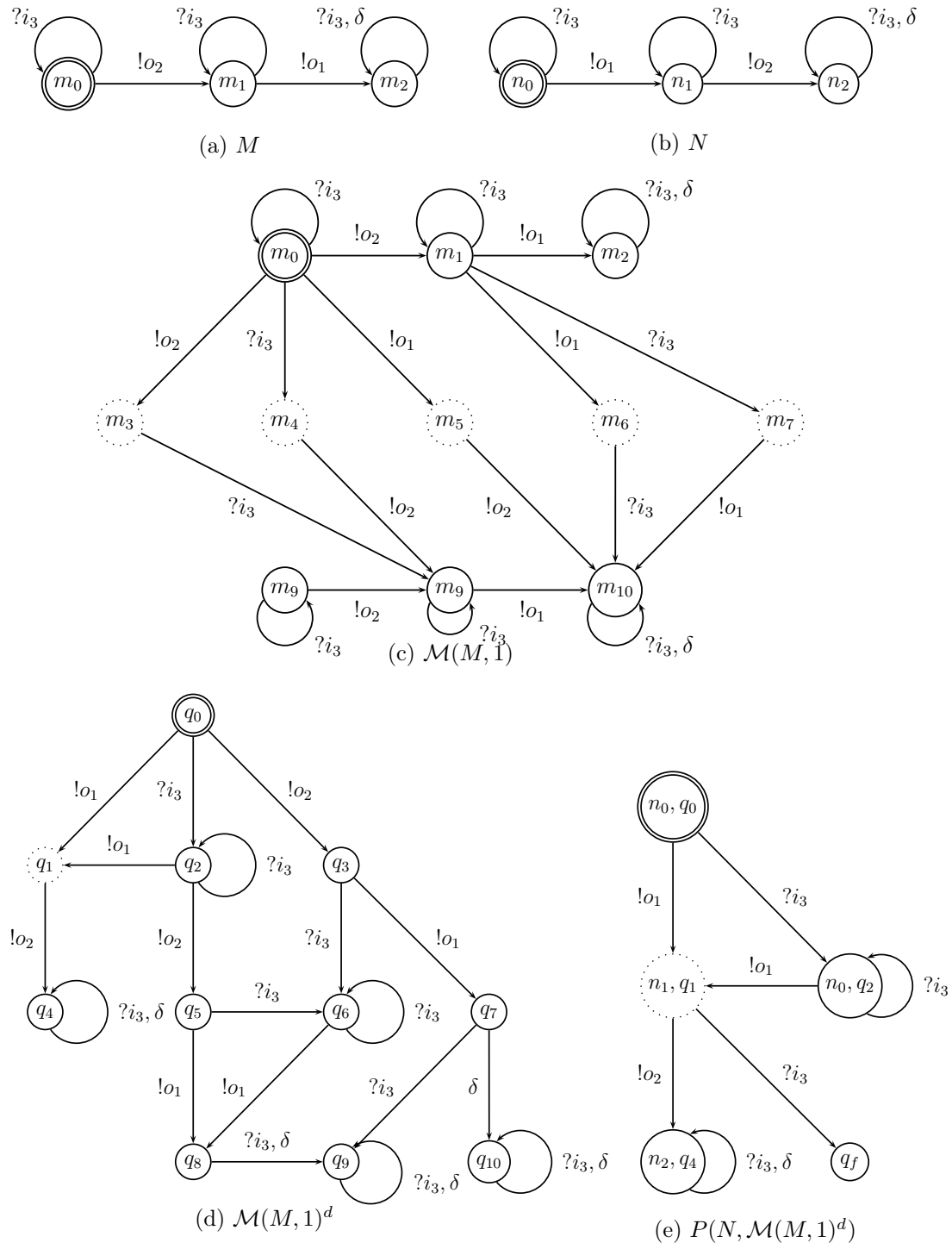
```
0 ?i_3 0
0 !o_2 1
1 ?i_3 1
1 !o_1 2
2 ?i_3 2
2 delta 2
@
```

Figure 3.3: Specification and implementation files

we can conclude that N $\mathbf{dioco}^{k}_{\mathbf{sw}}$ M.

In Figure 3.4 we present an example of the implementation. In this case, we have dealt with the *IOTSs* represented in Figure 3.4 (a) and (b), that correspond to the specification and the implementation, respectively. Both of them have been provided to the tool by means of the files depicted in Figure 3.3

Once the files have been parsed we apply the Algorithm 1 to obtain $\mathcal{M}(M, 1)$ shown in Figure 3.4 (c). As we can see, N is deterministic, so there is no need to determinize such automaton. However, $\mathcal{M}(M, 1)$ must be determinized to generate the automaton represented in Figure 3.4 (d). Finally, we produce the product machine corresponding to N and $\mathcal{M}(M, 1)^d$. In Figure 3.4 (e) we can observe that the fail state q_f can be reached from the initial state, therefore, we can assert that N does not conform M.



(a) M (b) N

(c) $\mathcal{M}(M, 1)$

(d) $\mathcal{M}(M, 1)^d$

(e) $P(N, \mathcal{M}(M, 1)^d)$

Figure 3.4: `IOTSs` M, N and automata $\mathcal{M}(M, 1)$, $P(N, \mathcal{M}(M, 1))$

Chapter 4

Conclusions

Testing is the main validation technique. Classically, testing was mainly a manual technique that strongly depended on the skills of the tester. However, it is now well recognized that "testing can be formal too" Gaudel (1995). The combination of formal methods, with a mathematical basis, and testing has allowed testers to (partially) automatize the tedious testing process. After an initial consolidation phase, where researchers were setting the foundations of formal testing, the work on testing started to consider more complex systems. For example, there has been a line of work devoted to testing systems with distributed interfaces.

In this thesis we have taken as an initial step previous work on testing distributed systems. Specifically, we have considered a formal testing framework to test systems with distributed interfaces where not all the allowed permutations are considered to be valid Hierons et al. (2018). We have shown that is it possible to check with a *product machine mechanism* whether N $\mathbf{dioco}^{k}_{\mathbf{sw}}$ M for a given $k \geq 0$ after expanding M into $\mathcal{M}(M, k)$ in order to recognize whether the implementation showed an unexpected behavior. We have fully implemented our contribution and we are able to decide how two systems relate. Another theoretical contribution of this thesis is that we have shown how the languages that these systems represent are able to be extended to be compared, that is, to decide whether they satisfy the properties to satisfy the **dioco** bounded relation with the considered distance.

In this thesis we have considered only one of the distances of the bounded relation defined in the original work (Hierons et al., 2018). As future work we plan to apply the ideas developed in this thesis to the other distance. Another

line of work is to consider *bounded* extensions of conformance relations based in **dioco**. Specifically, we have in mind previous work on probabilistic and timed extensions of dioco Hierons et al. (2014); Hierons y Núñez (2017).

Bibliography

Ammann, P. y Offutt, J. *Introduction to Software Testing*. Cambridge University Press, 2nd edición, 2017.

Binder, R. V., Legeard, B. y Kramer, A. Model-based testing: where does it stand? *Communications of the ACM*, vol. 58(2), páginas 52–56, 2015.

Bochmann, G. v., Haar, S., Jard, C. y Jourdan, G.-V. Testing systems specified as partial order input/output automata. En *Joint 20th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'08, and 8th Int. Workshop on Formal Approaches to Software Testing, FATES'08, LNCS 5047*, páginas 169–183. Springer, 2008.

Cacciari, L. y Rafiq, O. Controllability and observability in distributed testing. *Information and Software Technology*, vol. 41(11–12), páginas 767–780, 1999.

Cavalli, A. R., Higashino, T. y Núñez, M. A survey on formal active and passive testing with applications to the cloud. *Annales of Telecommunications*, vol. 70(3-4), páginas 85–93, 2015.

Dssouli, R. y Bochmann, G. v. Error detection with multiple observers. En *5th WG6.1 Int. Conf. on Protocol Specification, Testing and Verification, PSTV'85*, páginas 483–494. North-Holland, 1985.

Dssouli, R. y Bochmann, G. v. Conformance testing with multiple observers. En *6th WG6.1 Int. Conf. on Protocol Specification, Testing and Verification, PSTV'86*, páginas 217–229. North-Holland, 1986.

GAUDEL, M.-C. Testing can be formal, too! En *6th Int. Joint Conf. CAAP/-FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*, páginas 82–96. Springer, 1995.

HAAR, S., JARD, C. y JOURDAN, G.-V. Testing input/output partial order automata. En *Joint 19th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'07, and 7th Int. Workshop on Formal Approaches to Software Testing, FATES'07, LNCS 4581*, páginas 171–185. Springer, 2007.

HIERONS, R. M. Reaching and distinguishing states of distributed systems. *SIAM Journal on Computing*, vol. 39(8), páginas 3480–3500, 2010.

HIERONS, R. M. Oracles for distributed testing. *IEEE Transactions on Software Engineering*, vol. 38(3), páginas 629–641, 2012.

HIERONS, R. M. Verifying and comparing finite state machines for systems that have distributed interfaces. *IEEE Transactions on Computers*, vol. 62(8), páginas 1673–1683, 2013.

HIERONS, R. M. Generating complete controllable test suites for distributed testing. *IEEE Transactions on Software Engineering*, vol. 41(3), páginas 279–293, 2015.

HIERONS, R. M. A more precise implementation relation for distributed testing. *Computer Journal*, vol. 59(1), páginas 33–46, 2016.

HIERONS, R. M., BOGDANOV, K., BOWEN, J., CLEAVELAND, R., DERRICK, J., DICK, J., GHEORGHE, M., HARMAN, M., KAPOOR, K., KRAUSE, P., LUETTGEN, G., SIMONS, A., VILKOMIR, S., WOODWARD, M. y ZEDAN, H. Using formal specifications to support testing. *ACM Computing Surveys*, vol. 41(2), 2009.

HIERONS, R. M., MERAYO, M. G. y NÚÑEZ, M. Implementation relations for the distributed test architecture. En *Joint 20th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'08, and 8th Int. Workshop on Formal Approaches to Software Testing, FATES'08, LNCS 5047*, páginas 200–215. Springer, 2008.

HIERONS, R. M., MERAYO, M. G. y NÚÑEZ, M. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, vol. 25(1), páginas 35–62, 2012.

HIERONS, R. M., MERAYO, M. G. y NÚÑEZ, M. Timed implementation relations for the distributed test architecture. *Distributed Computing*, vol. 27(3), páginas 181–201, 2014.

HIERONS, R. M., MERAYO, M. G. y NÚÑEZ, M. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, vol. 67(2), páginas 522–537, 2018.

HIERONS, R. M. y NÚÑEZ, M. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, vol. 132, páginas 319–335, 2017.

HIERONS, R. M. y URAL, H. The effect of the distributed test architecture on the power of testing. *The Computer Journal*, vol. 51(4), páginas 497–510, 2008.

HOPCROFT, J. E., MOTWANI, R. y ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edición, 2006.

JARD, C., JÉRON, T., KAHLOUCHE, H. y VIHO, C. Towards automatic distribution of testers for distributed conformance testing. En *TC6 WG6.1 Joint Int. Conf. on Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE'98*, páginas 353–368. Kluwer Academic Publishers, 1998.

KHOUMSI, A. A temporal approach for testing distributed systems. *IEEE Transactions on Software Engineering*, vol. 28(11), páginas 1085–1103, 2002.

LUO, G., DSSOULI, R. y BOCHMANN, G. V. Generating synchronizable test sequences based on finite state machine with distributed ports. En *6th IFIP Workshop on Protocol Test Systems, IWPTS'93*, páginas 139–153. North-Holland, 1993.

MARINESCU, R., SECELEANU, C., GUEN, H. L. y PETTERSSON, P. *A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs*, vol. 98 de *Advances in Computers*, capítulo 3, páginas 89–140. Elsevier, 2015.

MAZURKIEWICZ, A. Traces, histories, graphs: Instances of a process monoid. En *11th Symposium on Mathematical Foundations of Computer Science, MFCS'84, LNCS 176*, páginas 115–133. Springer, 1984.

MYERS, G. J., SANDLER, C. y BADGETT, T. *The Art of Software Testing*. John Wiley & Sons, 3rd edición, 2011.

NGUYEN, H. N., ZAÏDI, F. y CAVALLI, A. R. A framework for distributed testing of timed composite systems. En *21st Asia-Pacific Software Engineering Conference, APSEC'14*, páginas 47–54. IEEE Computer Society, 2014.

PONCE DE LEÓN, H., HAAR, S. y LONGUET, D. Unfolding-based test selection for concurrent conformance. En *25th IFIP WG 6.1 Int. Conf. on Testing Software and Systems, ICTSS'13, LNCS 8254*, páginas 98–113. Springer, 2013.

PONCE DE LEÓN, H., HAAR, S. y LONGUET, D. Distributed testing of concurrent systems: Vector clocks to the rescue. En *11th Int. Colloquium on Theoretical Aspects of Computing, ICTAC'14, LNCS 8687*, páginas 369–387. Springer, 2014.

PONCE DE LEÓN, H., HAAR, S. y LONGUET, D. Model-based testing for concurrent systems: unfolding-based test selection. *International Journal on Software Tools for Technology Transfer*, vol. 18(3), páginas 305–318, 2016.

RAFIQ, O. y CACCIARI, L. Coordination algorithm for distributed testing. *The Journal of Supercomputing*, vol. 24(2), páginas 203–211, 2003.

SARIKAYA, B. y BOCHMANN, G. V. Synchronization and specification issues in protocol testing. *IEEE Transactions on Communications*, vol. 32, páginas 389–395, 1984.

SHAFIQUE, M. y LABICHE, Y. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, vol. 17(1), páginas 59–76, 2015.

URAL, H. y WILLIAMS, C. Constructing checking sequences for distributed testing. *Formal Aspects of Computing*, vol. 18(1), páginas 84–101, 2006.

WALTER, T., SCHIFERDECKER, I. y GRABOWSKI, J. Test architectures for distributed systems: State of the art and beyond. En *11th IFIP Workshop on Testing of Communicating Systems, IWTCS'98*, páginas 149–174. Kluwer Academic Publishers, 1998.

List of acronyms

IOTS: Input Output Transition System.

SUT: System Under Test.